Web Security and Privacy notes

Salvino Matteo

Contents

1 Email security

The mail system is made up of several components and aspects, it has a basic architecture and functioning, then the Internet community started to add some extensions (MIME) and enhance the basic functionalities adding security standards such as SPF or DKIM. Then, we will dedicate our time to understand if and how an email has been forged in order to trick the receiver in doing something wrong. Finally, we will spend some time talking about how to secure email, we will talk about PGP, which is possibly one of the first proposal to secure the mail system.

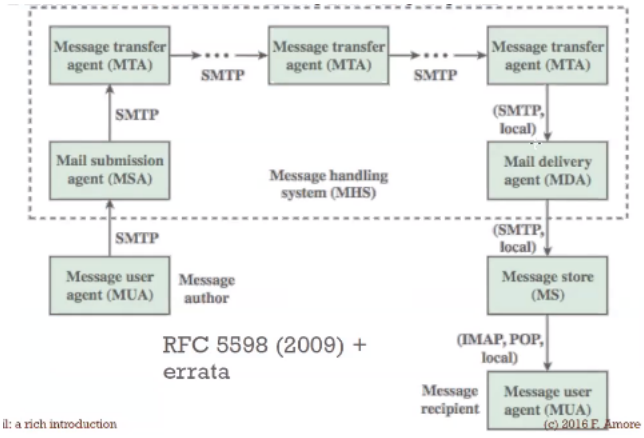
1.1 E-Mail architecture

The email service is something we are using today by access a very simplified client that hides a lot of complexity of the underlying system. In particular, the e-mail architecture is constituted by several components :

- **Message user agent (MUA)** : it's actually the client used by end user.
- **Mail submission agent (MSA)** : it sends the message to the recipient using a MTA. It interacts with the MUA to receive commands, store locally the information that are then passed to the MTA to create the real email.
- **Message transfer agent (MTA)** : it's what we typically call the mail or SMTP server. In general, we can have several MTA in order to reach the MTA of the email recipient. When the email arrives to such MTA, it recognize that it doesn't need to relay the message anymore and deliver the message to a MDA.
- **Mail delivery agent (MDA)** : it can possibly store the message on a MS. Typically, the MDA and MS are implemented by a single process.
- **Message store (MS)** : it's a location where the MDA can store the message received from the sender.
- **Message user agent (MUA)** : once the message is stored, it can be read or retrieved by the user using its client.

The interaction between MS and MUA is quite complex and it can be done through different approaches. The easiest case is the local approach, where

the user logs in on the machine that is hosting the message store and reads directly the message on the MS. The more common choice is to use a protocol to read email using a client that will fetch the email from the server and make possible for the user to read that email on a remote machine. This typically happens through different protocols such as **POP** and **IMAP**. With the first protocol, our client to the MS and say "give me a copy of all emails received from this date to this date". Next, it retrieves such emails and then they are deleted from the MS (there is also an option to leave the email there). The concept of IMAP is quite different; in fact the MS is designed to store emails. When we access the MS to read emails, we ask it to make a copy of a specific message we want to read. If we want to move emails or performs any other operation, we will sends commands to the MS, but the operation is performed on the MS. What we see with IMAP on our client, is somewhat a view of our message box on the MS. This is why the client need to use the protocol to synchronize its local view with respect to the reality that is given by the status of the MS. MTAs speaks between themselves using the SMTP protocol. Next, this protocol was extended to include some enhanced features such as encryption, which goes under the name of **ESMTP**. This extension is designed such that all software compatible with ESMTP will be also compatible with SMTP. SMTP exchanges messages using the message envelope separate from the message itself. The protocol also defines the way sender and receiver can be identified, typically using the standard email addresses. To have in mind the previous architecture we report the following picture.



In general, the way email are transferred on the Internet follows the concept of the **store-and-forward model**. This means that, every time a MTA needs to send a message to the next one, it first stores the message locally

and then forward the message to the next one waiting for a confirmation of the recipient (only at that point can delete the local message if needed). This enables one of the main characteristics of the system, which is the fact of being asynchronous. Typically an email is made up of three components :

- **message envelope** : it's used by MTAs to correctly relate the message from the sender MTA to the recipient MTA.
- **message header** : where we write who is sending the email, who is the recipient, i.e. they are meta information. In particular, it's structured into fields such as From, To, CC, Subject, Date and other information about the email. It's separated from the message body by a blank line.
- **message body** : it's the text of the message that we want to send, typically unstructured.

Each message has exactly one header, which is structured into fields. These fields are structured with name and value that are separated by a double column. There is also a possibility to define multi-line header fields. There are mandatory header fields such as :

- **From** : it's the sender email address.
- **Date** : it's the local time and date when the message was sent.
- **Message-ID** : it's an automatically generated field by the sender, to uniquely identify the message.
- **In-Reply-To** : it's a message-ID that represents a link to a previous message to which the current message is a response to.
- **To** : it's the recipient email address.

Other common header fields are the following :

- **Subject** : it's a brief summary of the topic of the message.
- **Bcc** : with this option we can add addresses to the SMTP delivery list but they will not appear in the message data, i.e. they are invisible to other recipients.
- **Cc** : it's similar to the previous field, but in this case all the addresses in the SMTP delivery list are listed in the message data.

- **Content-Type** : it's the field that allows the use of extensions like MIME type.
- **References** : it's similar to the In-Reply-To field, but contains a list of previous messages. It should be coherent with In-Reply-To field.
- **Reply-To** : it's mainly used to indicate to the client what to do if the user will click the reply button.
- **Sender** : it's the address of the actual sender acting on behalf of the author listed in the From field (list manager, secretary, etc).

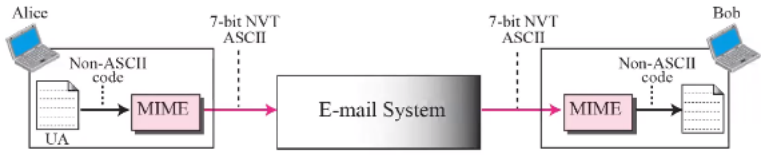
Together with the headers that we talked about, SMTP defines a lot of trace information that are included in the message as it's relayed toward the destination, which are also saved in the header using the following two fields :

- **Received** : it's populated every time the message is received by a MTA, with information about the MTA itself such as location, IP address, domain, etc. In this way we can rebuild the history of the message through the various relaying steps in its path toward the destination.
- **Return-Path** : it's a field defined by the final MTA.

1.2 MIME

Another standard often used in email system is MIME (Multipurpose Internet Mail Extensions), but it's also used in the web for other reasons. It's a standard that explain us how to encode complex content in more simpler containers. It allows to include : text in character sets other than ASCII, non-text attachments, message bodies with multiple parts and header information in non-ASCII character sets. MIME is one of the things that changed the usage of Internet in the last 20 years. Now, the question is how MIME works ? The idea is the following one : when the user want to send a message through the email system that contains some of information that are not in the ASCII character set, the client encodes this information using MIME in a transparent way. First it analyze the content that is provided by the user, checks how each part of the content should be encoded and set up a MIME compliant message with the correct encoding of each part. Then this MIME encoded message is fully expressed using only 7 bit ASCII encoding and can then be send via the email system. At the other end of the

email system when the message is received, the client of the receiver apply the opposite process, decoding the MIME content and correctly rendering the content of the message.



NVT = network virtual terminal

The important point is that the MIME encoding of the message body is completely transparent to the email system, because MIME provides a way to encode and decode this content at the endpoint, and everything is completely transparent to the email system since it never look at the message content. The kind of encoding used for MIME is defined through MIME headers which contains the version information, the type of the content that will be provided by the email body and other meta data such as encoding type, content-id and content-description. Some of the most common MIME types are *text/plain* for plain text data, *text/html* for html content, etc. If for example consider a jpeg image, this is not basic ASCII, how can we represent this richer content within a part of the body with the correct content type, but still sticking with the standard 7 bit ASCII ? For this reason, MIME defines several content transfer encoding methods : 7 bit, 8 bit, binary, base64 and quoted-printable encoding. Base64 is the standard way to encode non textual content in MIME messages. The idea is that through base64 we start from a binary content and obtain as target a string that is limited to $A-Z, a-z, 0-9, +, /$ character set. Typically the Non-ASCII data are divided in group of 8 bits, and combined in a certain number of 6 bit chunks. After that, each of them is interpreted as a number that is used as an index towards the 7 bit ASCII table. The good point is that having chunks of 6 bits we can range from 0 up to 63, which correctly maps to a 7 bit ASCII table. Considering that we are splitting and combining back the original stream of data, we may end up with some missing characters. So base64 encoded data typically may end with one or two symbols (they are equal symbols). The first advantage of this encoding scheme is for sure its simplicity to implement and apply to any kind of byte encoded content. Another good point is that, once we have the possible indexes that comes up from the various combinations of 6 bit that we use as chunk size, we can easily build a base64 converting table, such that an algorithm can just work

as an iteration on the byte stream that looks up characters that make up the encoded stream. The disadvantage is that we are just using a subset of the 7 bit ASCII character set, and this means that there is some space that is wasted in the encoding. The encoding is obviously not space efficient. Another option is called quoted-printable encoding, where any 8 bit byte value may be encoded with three characters : an equal sign followed by two hexadecimal digits representing the byte's numerical value; instead any non 8 bit byte values are ASCII chars from 33 to 126, excluding 61 (equal sign). It's extremely inefficient, because we encode 8 bits with 21 bits. However, its main advantage is that if for some reason the endpoint that receive the message is not able to decode quoted-printable encoding, the text is still mostly readable.

1.3 Unwanted email messages

Lets try to understand which are the weaknesses of these standard and how they are used to send unwanted email that typically comes with the name of **SPAM**. They are used to advertise any kind of merchandise. It's also used for a different goal such as performing some actions to take control of part or the full machine that the user is actually connected with. Another source of spam are called **e-mail chain letters**, with which induces you to fraud other people to get some benefit. Another characteristic of spam is the quality of the message itself. For example a message with a very low text quality, can be easily classified as spam. However, the low quality is something that is disappearing quite quickly. Naturally the spammers main goal is to spread the malware, in order to control computers for future attacks, steal sensitive information, and so on so forth.

1.4 Basic email nonalogue

We report a brief list of suggestions to follow when using the email system.

- Disable html message or, at least, disable download of remote images
- Don't click links, since we could redirect us to bad web sites containing malware
- Don't open unknown attachments since they may contain malware
- activate local anti-spam filter
- Don't participate with chain letters

- Protect and respect privacy of other recipients
- Even if non-Windows user, activate antivirus for protecting your recipients
- Don't provide your personal data
- Don't click "delete me" link.

1.5 SMTP extensions

We know that there are some pitfalls in the email system. For this reasons, there are other solutions that complements SMTP in order to provide some security features. In particular we will talk about three of them : **SPF**, **DKIM** and **VBR**.

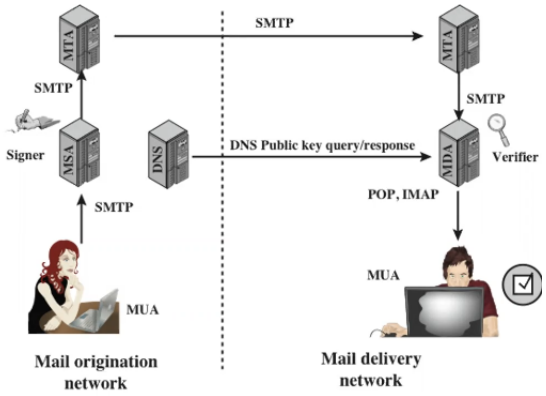
1.5.1 Sender Policy Framework

One big problems with emails is the fact that the body and the envelope of the email are somewhat separated entities. The body of the email is built in the client that creates the email, while the email envelope is forged on the server through the interaction between the client and the server. Furthermore, in the SMTP protocol no where is written that should be coherent some of the headers present in the email and what is written in the envelope (e.g. from field in the body from the same field in the envelope). SPF is a standard to solve this problem from a specific standpoint. In particular, SPF protects the sender address only in the envelope, it doesn't care about the from field in the email header. It allows the administration of the domain through which an email is sent, to define some policies about who could send emails through that server. So, receivers verifying the SPF records may reject message from unauthorized sources before receiving the body of the message. The nice point is that this will happen only looking at the email envelope; so is the server itself that can do this check and possibly reject the message, without the client even receiving the body. When the server accept the sender, it receives through the SMTP interaction the list of recipients and the message body. At that point it's expected to insert in the message header a field called Return-Path, in which it needs to save the sender address as is reported in the envelope. This means that this Return-Path may or may not correspond to the from header in the message. If the two are the same typically the message is fine; if they are different, SPF does not necessary mark the message as spam. We want to underline

that SPF doesn't enforce the correspondence between these headers; they should match, but not in all cases. Naturally SPF by doing checks only on the sender field in the envelope has some limitations : keep the SPF record update as companies change service providers and add mail streams is difficult. Another bad point is that given that SPF is not perfect, the check on SPF it's just one among several checks that email servers typically do to avoid spam to reach a target email. Another important point is that typically SPF breaks whenever a message is forwarded, because the forwarder address could not be whitelisted in the DNS text record and at that point the SPF check will fail. The last point is that SPF makes no check on the from header contained in a message that wrt SPF can be easily spoofed.

1.5.2 DKIM

Given that SPF only works on the email envelope, someone started to think about introducing a different solution to take care of the security of the message headers called **DomainKeys Identified Mail**. It specifies how some parts of the message can be cryptographically signed in order to avoid their content to be tampered by other people (MITM attack or non-intended source creates a fake message with false source information). The idea of DKIM is that when you receive a DKIM signed message, as a recipient you can verify the signature that is contained in the message by querying the domain dns server for the information needed to check that signature. This is important because the rational behind DKIM is that it links the content of the email that is signed, with an information that is obtainable through a query to the dns of the sender. So the idea is that the two will be coherent only if they originates from the same person (domain administrator) or people that the domain administrator allows to use this kind of feature. A classic deployment of DKIM works in the following way



First of all we have the origin of the email through a client. The client connects through SMTP with the server, but the call is intercepted by a **message signing authority (MSA)** that provides all the features to correctly sign the content of the message. Then the MTA through the SMTP protocol delivers the message to the endpoint to the server on the recipient side and this goes through a MDA, which checks through a query to the dns system if the signed content of the message is verified or not. If it's verified the MDA delivers the message to the end user. However, there is a practical problem when this protocol is widely deployed on the web. The email may pass through several servers, that typically may decide to add information in the headers of the email to keep track of what happened to the message on its path to the destination. Some of this changes can easily break the signature, because even if we just include the headers that are not expected to change in the path, some servers may include processing features that will alter the content of those headers. To avoid this kind of problem DKIM include a mechanism called **canonicalization**, through which it's possible to tell to the recipient if he needs to be strict in the checking of the signature of if he can perform a relaxed check. In this latter case, before applying the hash function to the headers, the recipient will for example make all the headers lowercase, remove all extra spaces, etc. Typically the relaxed checks says from easy rejection due to somewhat changes in the header that don't represent real mangling with the message content. Another detail that is important is represented by **selectors**. They allow to multiplex several private/public key pairs for the same domain, while providing a way for the recipient to check the correct one.

1.5.3 DMARC

In order to make SPF and DKIM work together to guarantee us that no one will make a wrong use of the available email services, we can use a standard called **Domain-based Message Authentication, Reporting, and Conformance (DMARC)**. It's interesting because it's not really a technical standard, but it's more or less a standard defining a process to integrate SPF and DKIM within a set of best practices that allow to make the best of them. In particular, DMARC allows organization to publicly declare policies through which they can define which kind of practices they put in place to guarantee email authentication. Furthermore, it provides instructions to the recipients to enforce these policies and report possible misuse. In particular, the domain owner publish its practices, he tell what to do when there is a failure with an authentication check and enables reporting.

How does DMARC works ?

1. The first thing you need to do in order to use DMARC is to publish a policy. The publishing of a policy is performed through your dns records and again it uses the TXT records embedding some information for DMARC associated with that specific domain. In particular, it uses a TXT record that starts with *_dmarc.* followed by the name of your domain.

2. When the recipient server receives an incoming email containing the DMARC headers, it performs first of all a dns query to check the policies of the sender. This query is based on the domain declared in the From field of the email. Next, it checks three aspects of the message : checks if the message's DKIM is valid or not, checks if the message come from IP addresses allowed by domain's SPF record and checks if the message headers show proper domain alignment. The domain alignment is an aspect that is specific of DMARC. It expands what SPF and DKIM actually already do, and it checks if various elements in the domain headers match the domain declared in the From field. In particular, for SPF it checks if the domain in the From field correspond to the domain indicated in the Return-Path field; for DKIM it checks if the domain in the From field correspond to the domain listed in the d field of the DKIM header.

3. With this information, the recipient server is ready to apply the sending domain's DMARC policy to decide whether accept, reject or flag the email message.

4. Finally, the recipient server will report the outcome to the sending domain owner. In particular, there are two formats of DMARC reports :

    - **Aggregate reports** : they represent statistical data on how many spoofed email for those domain has been received on a total of emails, or stuff like that. They are sent periodically, and not immediately sent at the reception of an email.
    - **Forensic reports** : they represent information about the checks the recipient server has performs and why they fails, but it has to include in the reports the original email that failed the checks. It allows the sender to perform some debugging for example to see if there is a problem with the DMARC configuration and it also

allow the owner of the sender domain to check if spoofing happens with specific patterns and possibly track down the origin of those spoofed emails and take some mitigations for that.

In particular a DMARC record has the following fields :

- **v** : it specifies the DMARC version to be used.
- **p** : it specifies which are the standard policies that needs to be applied to emails. It can assume one of the following three different choices : **none** for treat the mail the same as it would be without any DMARC validation, **quarantine** for accepting the mail, but place it somewhere other than the recipient's inbox, and **reject** in order to reject the message.
- **rua** : it's the mail address to which aggregate reports should be sent
- **ruf** : it's the mail address to which forensics reports should be sent
- **pct** : it specifies the percentage of mail to which the domain owner would like to have its policy applied (it's an optional parameter).

The outcomes from protocol checks are reported in the mail headers.

1.5.4 VBR

Another protocol that we want to talk about is called **Vouch By Reference (VBR)**. It takes a completely different approach that is still orthogonal to SPF and DKIM, and in fact it can be used together with these solutions. In particular, VBR allows the implementation of sender certification by third-party entities. The idea is that it allows providers to independently certify the reputation of senders by checking the domain name. When you send an email using VBR, you use the standard DKIM protocol to sign the email, and then include in the signature a VBR-info field, which includes several information such as the domain name that is going to be certified, which content of the message is allowed to deliver and a list of one or more vouching services, that is the domain names of the services that vouch for the sender for that kind of content. At the receiver side, after checking the correctness of the DKIM or SPF headers, the software may possibly double check the VBR info by performing dns queries on the domain name of the services included in the VBR-info field. In particular, it will look for entries of type TXT that have this structure *domain name._vouching service*. If VBR is checked at recipient side, the outcome of this check will be included in the Authentication Results field in the message header.

1.6 Basic email analysis

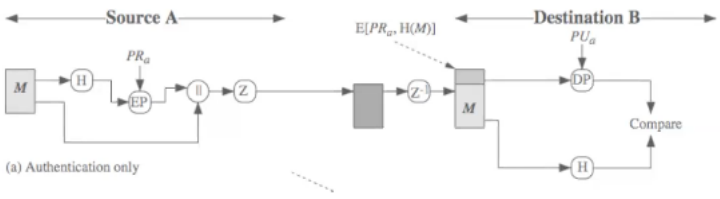
Email spoofing means altering some or majority of the elements that are present in the email header. The spoofer do that for hiding its real identity, since most of the content of spoofed emails is typically malicious in some form. How can the content of the email headers be spoofed ? At least for the sender address this is really simple, since SMTP doesn't include any authentication for senders. How do we check for spoofing ? Actually there are no deterministic approaches for doing that. For this reason, we can follow some guidelines that helps to understand if the message has been spoofed or not. A good starting point to do that is to analyze the complete message full header + body), checking the From, Return-Path, Reply-To and Received header fields. In particular, in order to implement a secure email service we need :

- **confidentiality** : protection of the content from disclosure
- **authentication** of the sender of the message
- **message integrity** : protection from alterations
- **non-repudiation** of origin : protection from denial by sender.

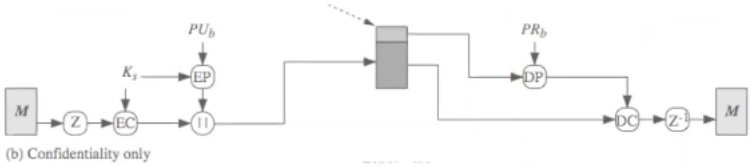
1.7 PGP

Pretty Good Privacy is an open protocol with proprietary and publicly open implementation. The basic point about PGP is that it was designed by incapsulating the usage of cryptographic primitive inside the email usage process. PGP is constituted by two main functionalities :

PGP Authentication When the sender creates the message, computes its hash and uses a private key to sign the hash. Next, the message and the signature are put together to form the signed message. On the recipient side, the message signature is decoded using the sender public key and the result is compared with the hash of the message itself. If the two hashes are coherent this means that the message has not been tampered.



PGP Confidentiality First the message is compressed and encrypted by the sender using a symmetric key called **session key**. Next, the session key is encrypted using RSA with the recipient public key. Next, the encrypted session key and encrypted message are sent together. The recipient takes the encrypted session key and decodes it using its own private key, obtaining the session key, that will be used to decode the original message.



Of course these functionalities can be used together (confidentiality & authentication). As previously mentioned, PGP uses a compression function to reduce the size of messages, typically before the encryption phase. In this way it's possible to store uncompressed message and signature for later verification, because the point is that compression algorithm may introduce some non determinism. When using PGP will have binary data to send, and it encodes such data into printable ASCII characters using Base64 encoding. Session keys that are used for encode messages are built randomly for each encryption pass, and the implementation is typically based on a 256 bit buffer which contains random bits. This buffer is updated every time you press a key interacting with PGP. It register the period of time between two keystrokes, plus the value of the keystroke itself, and use this information as a key to encrypt again the content of that buffer. In PGP you may possible own as a user more than one public/private keys pair, in order to send different kind of messages or to interact with a different group of people. To identify which couple has been used to encrypt a message, the message is sent including the public key plus the ID of the key identifier of that couple of key. The key identifier is simply extracted from the least significant 64 bits of the public key. In PGP implementation each user maintains through its client two **key rings** : the public key ring contains all the public keys of other PGP users known to this user, indexed by the key ID, the private key rings contains the public/private key pairs used by the current user and they are indexed by the key ID and encrypted using some passphrase. Now the question is, how do we manage these keys ? In PGP every user represent its own Certification Authority (CA), which is able to create all the public/private key pairs needed. The main disadvantage is that this removes the so called assumption on the presence of a trusted third party, that makes this approach usable for example for signing digital

certificates. Instead, PGP rely on the concept of **web of trust**. The idea is that when you start using PGP, you need to get the recipients public keys, and then start to trust these recipients and they will trust you. This mechanism will iteratively creates a web of trust, where keys brings with them a sort of certificate of trust that says that those keys have been trusted by other users and this should increase the possibility that they are also trusted by someone else that have never seen before those keys. For this reason, key rings includes trust indicators in their data structure. This web of trust somewhat is a vision that looks at reality where the trust is something that emerges from a decentralized fault-tolerance web of confidence.

1.8 ARC

The **Authenticated Received Chain (ARC)** protocol proposes a solution to overcome the problems that we may have using SPF and DKIM. In particular, instead of enforcing security only between the sender and receiver, it allows to completely track security checks through the full chain of relays that are encountered while an email is in transit. The reasons why messages can fail DMARC policies checks if they are strict and the message are passed through some indirect mail flow such as mailing list, or caused by attachments removal, etc. To overcome these problems people started to think about DMARC trying to sketch out which are the main design choices for the definition of this new protocol. In particular, the main design decisions are the following :

- the originator of the message doesn't need to make any change to the message itself
- convey Authentication-Results content intact from the first ARC intermediary forward
- allow for multiple hops in the indirect mail flow
- ARC headers can be verified at each hop
- work at Internet scale
- we need to make ARC as independent as possible from DMARC.

The idea of ARC on the recipient side is that the receiving endpoint should perform the standard DMARC checks. But if a failure is result of these checks, then optionally the receiver may possibly look for the presence of ARC headers, and use them to perform a more extensive check. If this

extensive check give us a negative result then the server may possibly decide to locally override for that message the final decision for the DMARC policy. How does the ARC headers checks actually works ? Through the ARC header the final recipient need to be able to look at the value of the original Authentication-Results field as it was created by the first hop in the mail forwarding chain, and then checks the authenticity of all the intermediate steps that have been traveled through by the email from that first step up to the recipient point. So, the recipient will be able to validate the entire chain. Thus an intact ARC chain provides to the recipient the following information : it gives DKIM, DMARC and SPF results as they have been seen by the first hop in the mail forwarding chain. Then it provides signatures that show that these results were actually not tampered with someone else. Finally, signatures from participating intermediaries can be linked to their domain name to prove that they are actually what they pretend to be. However, this doesn't prevent intermediaries to alter the message content. The main point of ARC is that such alterations can be done only with attribution, with which we are able to identify who did such changes in the chain. ARC doesn't provide us a solution for any problem. For example ARC doesn't tell us how much trustable is the message sender or the intermediaries, it only provides us information of what they did with that message. It says nothing about the content of the message and what has been added by the intermediaries to the content itself. ARC introduces the following header fields :

- **ARC-Authentication-Results (AAR)** : it's an archived copy of the Authentication-Results as it has been seen by the first hop. Each intermediary will performs the checks locally. They will give rise to several ARR headers that will be relayed up to the final recipient. Given that they need to be ordered for the final recipient to correctly interpret their content and the evolution of these checks through out the full mail forwarding chain, the AAR header also include a special tag i that contains a sequence number. The only Authentication-Results field that will not be archived in AAR header will be the one that is calculated on the final hop, i.e. the one computer by the recipient.

- **ARC-Seal (AS)** : it's a header that includes a few tags that are needed to understand how ARC should work, and a DKIM style signature of any preceding headers. It's built like an incremental seal that provide a proof of the fact that all the ARC headers including AMS and AAR headers that are included in a message have been correctly

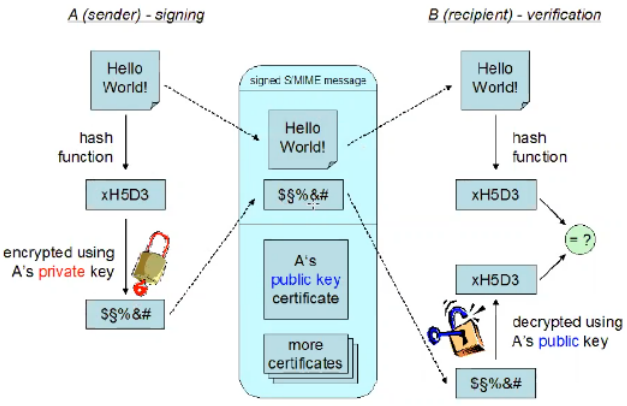
created and have not been altered during the travel of the message through out the forwarding chain.

- **ARC-Message-Signature (AMS)** : it's a modified DKIM signature.

The order of insertion is that first we have the AAR which are created with a simple copy of the Authentication-Results field, then the AMS and finally the AS. To summarize the ARC benefits for sender/intermediary are the following : it allows intermediaries to continue and/or resume traditional From semantics, message modifications, allows more senders to adopt strict DMARC policies, and improves overall deliverability. While the ARC benefits for the receiver are the following : less stress for enforcing DMARC policies, allows more mailbox providers to publish strict DMARC policies on their customer-facing domains and we have more data for reputation systems.

1.9 S/MIME

The authors of **S/MIME** protocol took all the approaches defined in PGP for signing and encrypting emails, and simply integrated them with a standard digital certificate based approach to trust. S/MIME is a widely used protocol for sending and receiving emails in a secure fashion, and its name comes from the fact that it somewhat extend the standard MIME protocol for encapsulating different kind of content in a single document, and it adds MIME types that are used on purpose to handle security mail content. Furthermore, it also adds some indication on how it implements email integrity and confidentiality. We can see that email signatures are handled exactly as in PGP



The original message is passed through a hash function in order to provide a digest, which in turn it's signed using the sender private key. Then the original message and the signature are embedded in a S/MIME message. Differently from PGP, the message typically includes the public certificate of the sender and may possibly include other certificates that are needed to correctly enforce the trust upon this public key certificate. On the recipient side, we have the classic approach used in PGP, i.e. the recipient computes the digest of the original message, decrypt the signature using the sender public key extracted from the sender public key certificate, and checks if the result matches the digest previously computed. When we move to encryption, it's performed exactly in the same functional way, but using X.509 standard certificates. A random session key is chosen on sender side, it's encrypted using the recipient public key and then the original message is encrypted using that session key. Both the encrypted message and encrypted session key are encapsulated in a S/MIME message plus the sender public key certificate. Then on the recipient side, the opposite approach is used to decrypt the message as in PGP but for the presence of certificates. The real strong difference between PGP and S/MIME lies in how the identity of sender and recipient is handled by the protocol.

1.10 How PEC works

To implement a mechanism on top of the email service to include beyond security functionalities also tracing functionalities that allows to trace what happens to the emails and make these traces visible both to the sender and receiver. A possible approach that does exactly this goes under the name of **Posta Elettronica Certificata (PEC)**. It works as follow : the sender wants to send an email to the receiver but he wants also to have a confirmation about the steps performed to manage this email in its travel from himself to the receiver. It's mainly based on the usage of S/MIME, plus a set of procedures that are needed to create certified timestamps about what is happening to the email. In particular, the sender creates an email and ask its service provider to deliver such email to the receiver. The provider at this point only enforce the authentication from the sender, i.e. it checks if the sender is actually what he's pretending to be and the content of the headers in the email are coherent with the digital identity of the sender. Once all these checks are ok, the provider sends back to the sender a certificate, which contains information about such checks and that the email has been taken in custody by itself. Next, the provider will take the content of the email and encapsulate that email in an envelope that is signed by itself.

The envelope is then sent from the sender provider to the receiver provider through a reception point. The receiver provider checks if the signatures in the envelope are correct, and if it doesn't see any problem with the email, it will take into custody the email, and will send to the original sender a new certificate a new certificate that attest the fact that he received the email from the sender provider with a specific timestamp and that he's now going to deliver the message to the receiver inbox. At this point, the receiver provider access the receiver inbox and place the message in it. When this happens, a new message will be sent to the sender saying now the message is in the receiver inbox. In this way we have an email that is signed with a signed timestamp, and they are legally usable to attest the fact that an email was correctly delivered to a given recipient.

2 Introduction to Web Security

In the recent 15 years there was a strong increase in attacks that targeted multi-tier applications deployed on the Web. This increase follows in general two trends : it follows the trend of the growth of what we call Web 2.0 applications, where the content is adapted to the specific needs of who is reading that and where web sites move from a situation in which they were mostly constituted by static content to a new reality where they represent full fledged applications. Naturally, protecting these kind of systems is way more complex than protecting a single standalone application on a pc, because simply there are way more levels of freedom that make these applications more dynamic, but on the other side make the job of security operators more complex. Notice that in this kind of scenario the critical components that needs to be protected are typically placed on the server side (DBMS server, authentication server, etc).

2.1 OWASP

This topic became so pervasive during the last 15 years that a few initiative arose to try to provide a comprehensive view of the problem and suggesting meaningful and still simple practical solutions. Among the various projects, one of the most interesting is called **Open Web Application Security Project (OWASP)**. It provides some best practices to be adapted in order to make systems more secure. The basic principles suggested by OWASP for securing complex web application are the following :

- Apply **defence in depth** : it refers to the fact that the best practices linked to border-based defences are still good, as long as we define fine grained borders also within your premises. In other words, apply defense in depth means applying layered security both horizontally (working for different abstraction levels in our organization) and vertically for functions. This layered security allows us to create a lot more borders that we can control and enforce. Its main advantage is that if some adversary manage to overcome some of our defense measures at one of our borders, then there is quite a large probability that he will be able to gain an advantage point in our premises that is still not enough for him to reach its final intended goal. We should not rely on an unique defense.

- Use a **positive security model** : typically it's depicted as a choice between two opposite sides using a negative security model or a pos-

itive security model. This latter one is historically the one that need be used and it's based on the usage of whitelist. The idea is that, whenever we apply this approach, we need to deny everything and then making exceptions by whitelisting them.

- **Fail securily** : when a system for some reason fails, then is important to guarantee that it fails securily, i.e. errors are handled such that they do not creates security issues. From this point of view we can recognize two cases :
  - **type 1 errors** : they are errors that happens during the processing of a security control. Obviously they should disallow the operation.
  - **type 2 errors** : they are errors that are exceptions in code that are not part of a security control, but given how this code is designed, they may affect the way a security control is performed.
- Run with **least privileges** : with this approach we give to accounts the minimum amount of privilege that is necessary to perform a given operation. Of course, this privilege should last just for the time needed to perform such operation.
- Avoid **security by obscurity** : it's a strategy where the idea is that if you want to encode things what you should keep secret is just the key used to encode, but then algorithm should be free, open and known to everyone, because it's not by hiding things that you will make those things secure.
- Keep security **simple** : complexity could give us a false sense of security, but actually it's something that works against us. The more complex is a security system the easier it will be for it to contain errors, bugs, misconfiguration or bad design decisions.
- **Detect** intrusions : it's fundamental to log security-relevant events and to ensure that they are periodically monitored, in order to act as soon as possible to limit and mitigate the intrusions impact.
- **Don't trust infrastructure** : this means that understanding our infrastructure in details is fundamental to avoid errors and pitfalls that may give arise to incidents.

- **Don't trust services** : typically most applications are constituted by a set of open source libraries that works in a coordinated manner, but we cannot expect that they are secure.

- Establish **secure defaults** : we need to make sure that, even for people that are unaware of security pitfalls and threats, using our assets is something that can be in a secure fashion. We may possibly give the opportunity to make informed choices how secure is the usage of our assets and resources.

This practices works for a wide variety of potential risks on server side, client side, but also for ones that comes from the network.

2.1.1 Server side risks

They are typically the most effective ones and they are related to different objectives that adversary may have when they plan how to attack the servers we are running. Their goals are typically the following : **data stealing** (i.e. data breaches), **remote command execution** where the attacker is trying to deploy and execute on the target machine, code that is written by himself for various purposes, collecting further data for setting up more complex intrusions or attack the machine through a DOS to interrupt the services provided by such machine. We are dealing with the security of applications that offers services to end users, and it's important that these services are offered in a safe way such that the users will be able to access functionalities offered by services as they have been designed by the system designers. An attacker to subvert the functioning of the server is to modify the application itself either to hide behavior that not directly visible to the users like steal their information or possibly to push the users to do some actions that would never do in a normal scenario. For this reason the Web server that run the application plays a fundamental role. In particular, it's very important that their configuration is appropriately managed by system administrator to make sure that content and applications are executed in a safe environment. From this point of view there are few elements that we need to take into account :

- **document root** : it's the folder where the web server will look for content to be served against users requests. Typically inside this folder we will find documents and files that represent the starting point to serve the content towards the users. Clearly the document root is another delicate point in the configuration, since its content needs to

be reachable from users with privileges such that to avoid that they can possibly execute code that they provide inside this directory.

- **server root** : it's the directory where the files and executables needed to correctly execute the web server are actually located. It typically contains few scripts, logs and configuration files. It's fundamental to protect this directory, since for example the logs files may contain sensitive information such as session key, and an attacker may try to steal it in order to replicate a valid session. Furthermore, this directory has a lot of space and subdirectory that can be accessed with different level of rights in order to correctly execute the server. However, an attacker may leverage such space to inject its own executable and make it run when the server starts, and he could leverage it for example to receive commands from a remote server.

Most of these problems boils down setting up these directories with the correct file permissions wrt the configuration of the web server. In particular, we need to take into account that document root will contains mainly documents that just need to be read and that server root will contains files that needs to be execute or documents that shouldn't be accessible from normal users. A common approach used to secure directory and user access rights is by defining specific user name and group for managing the access rights to these two directories (e.g. www & wwwgroup). Typically, the server root will be the home directory of www user. Instead, the wwwgroup is used to make sure content editors will be able to access directories with the correct rights to write new documents inside the document root and its subfolders. For the server root typically only the www user should have read/write/execute privileges everywhere within such folder and its subfolders. Users part of the wwwgroup may have read and execute access everywhere, but write permissions only on content that needs to be updated. Notice that the web server should run with a user called *nobody*, which has minimal access rights on the target system. This means that clients accessing the web server will have exactly those rights. For this reason, it's important that the document root will give permissions for reading and executing to other users different from www user and the ones in wwwgroup. In some cases, we may want to implement through the web server selective access to the content of the document root depending on the browser's IP or authentication. However giving permissions to all users to be able to read those documents is not a good idea, because all local users will be allowed to access that content. In this case, we may possibly reconfigure the server to run with a different user from *nobody*, provided to that user minimal privileges

and belongs to the wwwgroup. In that case, we need to make sure that such group has a restricted access with write rights to specific sections where we know that the users will not be able to do any damage. If for some reason, the web server runs with a user different from *nobody*, we need to make sure that no log directory is writable for that user id, otherwise it may possibly be used by an attacker to write code to be execute or to gain access to protected data like the passwd file, by simply creating a symbolic link within the log directory and then read that symbolic link. In the typical scenario, the web server is started as a daemon with user id root, and this is needed to allow the server to stay listening on reserved port such as 80 or 443, and to write log files in a protected place. Naturally, if someone connects to the web server, this latter accept the connection and spawn a new child process that will be executed with limited privileges (typically with user *nobody*). This is a robust scenario because it allows the web server to have full control of the machine for correctly serving the incoming requests and make sure that such requests can be handled by low privileges processes such that if any of these processes is subverted the impact of the compromisation is as small as possible. If we allow to run the child processes with root privileges, then everything can be accessed by the users and possibly the attacker can use the vulnerabilities in the code executed by the web server to make damage like only an administrator can possibly do. Another scenario, is where not even the parent process has root privileges. However, in such case the web server won't be able to open port 80 and neither other well-known port.

Some optional services that may be offered by web server and that represent another source of concern of security risks. In particular, most web servers includes the following services :

- **Directory listing** : it's a service that allows the users to get a directory listing. This is a strong problem, because we may end up stuff in document root that represent backup files, temporary directories and files. Notice that just disabling this service, doesn't prevent attackers from accessing these files.

- **Symbolic link following** : it's a service that allows requests to follow symbolic link. It represent a possible security risk, because if in the document root is present a symbolic link to an external file, then a client that request to access to such link will read the content of the linked file. A possible solution is to use some configuration opportunities provided by the web service (e.g. aliases) to extend the

document tree beyond the limits of the document root. In this way the configuration is more explicit wrt a symbolic link.

- **Server side include** : it's a simple way to include in static content some small snippets of dynamic data, that was provided by incorporating an external file or including in the html static content the result of the execution of an external process. In particular, among the various server side include directive there was the *exec* directive, that allows the execution of external scripts or shell commands, took the outcome and include it in the html file before serving it to the client. This is a common security risk because it allows, if not appropriately controlled, to make the attacker run more or less any potential script in the system and serve whichever content he wants. Today most of the functionalities that were formerly proposed through server side include are implemented by more complex full fledged server side scripting techniques based on various languages such as PHP, Python, etc.

- **User-maintained dirs** : it's a service that allows users to automatically add to document root personal portions of file system. This is potentially a strong breach in security, because in the end server administrators typically are not able to fully control what users actually do.

There are also some cases where content creator are given the possibility to access some sections of the document root through another service like a ftp server, that was used to update the content of the web server. The problem is that now we need to guarantee a coherent configuration of both servers in order to avoid potential vulnerabilities to arise from their possible connections. An approach that is sometimes used to enforce a bit more security on web servers is to use *chroot* command, which creates a limited execution environment with root the provided path. The problem with this kind of approach is that processes will only see things that are visible through the new root. In this environment we can avoid to include any interpreters or sensitive data that can potentially be used by an attacker to execute code or steal information. Another important approach that is used to secure the environment that is around the web server, is to isolate the web server wrt other potentially sensitive assets like hosts and networks that need to be better protected. This approach stems from the idea that the web server is something that needs to be accessed by anonymous users. While other parts of our networks don't have this kind of requirement. This approach is realized in such a way that, even if the web server is compromised, the

attacker will not be able to get into the entire network. Typically the server is placed in an isolated network area called **Demilitarized zone (DMZ)**, which controls the access from the internal network to the web server, and forbids the access from the web server to the internal network. The implementation of this kind of network segmentation is done using an appropriate use of the firewall, which may give arise to the following approaches :

- **Dual homed host** : in this case the firewall is equipped with two independent network interfaces, one is attested on the Internet and the other one is attested on private network, the routing between two interfaces is disabled by default, and packets are allowed to flow between the Internet and the private network only by passing through an appropriate application that perform in general application level filtering. In this case we would place the web server between the dual homed host and the Internet, but the problem is that this kind of configuration is not flexible, because any exception requires an appropriate rule to be written at application level within the dual homed host to allow packet flows and with large networks this hardly scale.

- **Screened host** : this approach is based on a router called **screening router**, which is configured to block any connection from the Internet to a protected network, but any of these connection attempts is redirected to a special host called **bastion host**. It's the only host present in the protected network that allows to access and receive connections from the Internet, and in practice it acts like a proxy wrt any connection that goes from the Internet to the protected network and viceversa. Clearly, the bastion host needs to be strongly protected (hardened setup). It's often realized through secure operating system that are standard operating system configured to offer as less services as possible such that they will not be exploitable by attackers. Every activity happening on the bastion host is immediately logged and placed on a secure log service such to be continuously monitored for possible traces of incidents.

Another important aspect to take into account while protecting server is to continuously monitor what is happening in that server. The server typically exports information in a detailed log that contains all information about which kind of actions are performed. The idea is that if the server is compromised, the log can be modified by the attacker, in order to hide its activities. In this case it makes sense to equip the server with a host-based intrusion detection system, which is a software that runs together side by

side with the server on the same host, and continuously check and analyze what the server is doing. They are able to intercept some suspicious activity and in such case they may possible raise an alarm or take some remediation action such as terminating that specific process and so on so forth.

2.1.2 Client side risks

From the client side, most of the risks are related to the browser. They are a good target for attackers because they may possibly be used to run the user code to call whichever effect may be the interest of the attacker such as browser crash, damage the user system, etc. In particular, the violation of end user data is a compelling scenario, since these data can be used later to perform identity theft attacks. In general, the attacker may use the vulnerabilities present in the user browser to attack the user itself by compromising a system as soon as a system visit the website, possibly delivering on the machine code that represent malicious software like trojans or spywares. An example of trojan is the following : we visit a website which download and execute on the web browser some client side code that exploit a local vulnerability. An example of spyware is the following : it's a software that sell itself for performing some legal action on the system, but they are written in a tricky and unclear way in such a way to push the user to accept the proposed conditions, making him subject to some type of legal data stealing from its machine. Today there is a security policy that is enforced by all browsers called **Same origin**. The idea of such policy is the following : if you visit a website and you exchange some data with that website, another website cannot access that data. The term origin is defined using the domain name, application layer protocol and port number of html document running the script. Two resources are considered to be of the same origin if and only if all these values are exactly the same. Why this policy is so important ? Because it allows the browser to implement easily a strong security policy that somewhat create some clear and unavoidable barriers between distinct applications and avoid these applications to inadvertently exchange information.

Cookies They are simple pieces of information that a website can store on a visitor web browser. It's a way to overcome the main limitation of http protocol, which is the fact that it's a stateless protocol, i.e. it doesn't maintain any conversational support. For this reason the standard builders decided to introduce the cookies as a tool that can be used for several distinct things, and among these things is to implement a simple mechanism to

track user sessions. The idea is that every time a web browser request a web page, the web server will provide the content that has been requested, but it may also include some headers that includes cookies. These cookies are simply small bits of information. Typically to track session the web server will include a session ID (a pseudorandom number which is with high probability unique and that will uniquely represent that user every time the user interacts that web server during that session).

Once the cookies are stored in the web browser, every time the browser will make a new request to the web server, it will send together with the request the cookies. On the web browser, cookies are stored locally in a dedicated storage and it will make sure that they are send back to the web server only when the user is visiting the web server for which that specific cookies has been designed. In particular, each cookie will contain a few components such as name and value (mandatory), plus other optional components such as expiry, path, domain and need for a secure connection. Typically, we can find two kinds of cookies :

- **session cookies** : they are removed when the browser quits (they do not have an expiry date)
- **persistent cookies** : they are cookies that has an expiry date which remains on the client until such date. Typically, they are not kept in memory, but they are also saved somewhere on disk. Their typical usage is for example to automatically authenticate again the user as long as he tries to log in in a near future.

Another interesting way to look at cookies is by differentiating between :

- **first-party cookies** : they are provided by a website that we target directly by making a request through our web browser.

- **third-party cookies** : they are injected by other websites that we don't have directly visited or we didn't do that explicitly. This may happen for example if we visit a website A, which provides the content we are requesting and possibly also includes some first party cookies. Then our browser analyzed the content that has been received by A and while rendering the web page it encounters an element in the page that requires further content be fetched by another website B. In order to do that the browser will open a new connection toward domain B and through http request ask domain B to provide the resource that is needed to correct render the page. While providing that resource website B has the opportunity to send back to the browser a cookie (this is consider a third-party cookie). This is typically a trick that is widely used by marketing companies to implement large scale of tracking of users preferences. In fact, the tracking is obtained by embedding in a web page an advertisement that will keep track for example of how frequently the user will visit such web page. However, there is still a problem, because the same origin policy doesn't allow website A to exchange information with website B. In the standard scenario the website A will make a request to website B, but will not include the cookie that are stored by it, because this will violate the same origin policy. What B can possibly do, is to store on our browser information about our identity from its standpoint. The two identities on A and B are not obviously linked. However these two identities needs to be linked in some way in order to allow an exchange of tracking information between A and B for a particular user. How can they do that without violating the same origin policy ? They can use the so called **cookie syncing** approach. The idea is the following : when we ask A to provide a content at the beginning of a session and provide our user ID, A before serving us the content will provide us with a redirect http response that will point us to website B. In this request A will include both its ID and our user ID as its known by A. Our web browser will bring us to B sending these information plus the cookie for such website. At this point B will match our local user ID with our remote user ID on website A. Now, for B is enough to redirect us again to A with an information that will tell website A, "ok, i've synched the ID, you can proceed with serving the original content". How much can we protect ourselves from this kind of information leakage ? The most extreme solution is to use an ad-hoc solution that completely anonymize our usage of the web (e.g. TOR). Another approach is at least protect our point of access to the web

by using a VPN. The usage of cookies for tracking user preferences is more and more endangered on one side by technical advancements like those introduced in the browsers and on the other side by regulators that tend to limit the freedom of tracking services to do whatever they want to profile users. However companies continuously look for new solutions to overcome these limitations. An example are the **super cookies**, which are information still stored in our computer by non-traditional methods. They are persistent, but the fact that they are not real cookies, they tend to be ignored by both regulations and technical means to get rid of them.

The risks coming from the network are linked mainly to actions that the attackers may perform by interposing itself between the two endpoints that are communicating. The basic attack is linked to **eavesdropping** to capture data that are exchanged between the browser and the server, and possibly make use of these information.

2.2 HTTP authentication

For accessing a lot of online services we need to authenticate ourselves. We have a physical identity, but then we act in a virtual world using a digital identity (a representation of ourselves in the digital world). When we asked to authenticate by a service, the service will challenge us to prove the fact that our physical identity can be represented by a specific digital identity while using that service. Authenticating means proving that there is a match between our physical identity as user that is accessing that service and the digital identity that we will use for accessing that service. There are several ways to authenticate users. We will start talking about **HTTP authentication**. It's a protocol performed by landing on a web page that requires authentication, where the web server will fires up the request for authentication and the browser will answer asking us to provide a digital identity in the form of a username and password that confirms the link between that digital identity and ourselves. The HTTP authentication is typically configured on the server side by configuring in the proper way the web server. It implements two different challenge-response protocols :

- **Basic authentication** : assume that we perform a request for a resource on a web server and the server is configured in such a way to require authentication in order to access that resource including in the response header the WWW-Authenticate header that will include at least the form of authentication required plus realm=string.

The realm concept is used to contextualize the authentication to a specific security realm on both the client and server side. When the client browser receives the 401 error message since the client didn't performed any authentication yet, it will show to the user the classic popup asking for username and password. The username and password will be send back by the client towards the server replicating again the same request, but adding the Authorization header that will contains the type of authentication used plus a string. This string contains an encoded version of the username and password (Base64 encoding of username:password string). When receiving this request with the header, the server will take the content of the username and password from such string, checks the credentials locally, and if everything is ok, it will return the status code 200 and serves the request, otherwise it will send back the 403 error code. We want to underline that this schema doesn't use any encryption method, so it's very easy to decode such string in order to retrieve the user credentials.

- **Digest authentication** : It's an authentication method slightly more secure than basic authentication. In this case, the server challenges the client using a nonce (together with www-Authentication and realm header). Then the browser will again ask the user to provide username and password, and it will the information using again Authenticate header, with the username in plaintext, the nonce received from the server, the realm, the uri of the service that needs to be accessed, a response (which is a cryptographically hashed version of the username and password) and the algorithm used to compute the response. The client to compute the response uses a predefined a cryptographically secure hashing function such as SHA256, and then simply first applies the hash to the string username:realm:password, then it applies the hash to method:digestURI and finally will it will hash the string constituted by the first hash computed, followed by :nonce:, and the second hash computed. The server replicates the same approach and then will check coherence of the calculated hash with the response that was sent by the client.

Another type of authentication is the **form-based authentication**, which is a simple way include all those authentication methods that rely on the user sending credentials through some kind of html form. It basic idea is that the user is redirected to a specific web page on the web application that he's using. This web page provides the basic form where it request the user to provide its credentials. Naturally, the requests to this

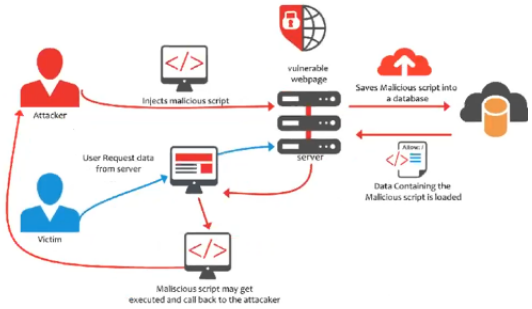
page needs to be protected through TLS. So the user filling the fields with the username and password, click the button to submit the credentials that are sent through the secure channel towards the web server, which it will validates the user identity. Exactly like the previous methods, it's designed to overcome overcome the limitations imposed by the stateless nature of HTTP. By asking user to authenticate, the web application fulfill two different goals at the same time : on one side it creates a persistent session for the user, and on the other side it associate this session to a specific virtual identity. What the server typically does upon authenticating the server, it sets through a cookie a session ID for the authenticated user. This session ID needs to be protected, because it represents the authenticated user for the duration of its usage of the website. So, from this standpoint, the session ID represents the identity of a logged in user. This means that if someone is able to sniff and steal that session ID, he may possibly to impersonate the user by sending new requests and independent from the ones of the real user, sending also the session ID. In this way the attacker is able to fully impersonate the user, unless the server implements some techniques to avoid this kind of attack. This attack goes under the name of **session hijacking** or **cookie hijacking**. So keeping the client-server channel secure by encrypting it by TLS is a fundamental requirement to implement form-based authentication and the subsequent secure session IDs secured through out the whole session. However, this is not the only way session hijacking can be performed. Indeed, if the attacker knows that the client-server channel is encrypted through TLS, he will look for alternative solutions to steal the session ID (attacking the server or the client). The security of the browser is typically linked to two aspects that are tightly coupled with the user that is actually using that browser (browser bad configuration or when the user give the permissions to the attacker to be attacked without understanding that). Fairly more frequent are attacks that target the server side. This kind of attacks typically target the front-end of the web application. They may have different purposes : target the web application to subvert the functioning of the web application, to steal data from the web application, attack specific users. Now, we want to talk about **unvalidated input** which is a typical software design flaw that give rise to different vulnerabilities that can be exploited with different kinds of attacks.

Unvalidated input Typically all web applications rely on the standard HTTP request-response protocol to provide their functionalities. If the client is an attacker he may try to subvert the application functioning by tamper-

ing with the content of the data streams exchanged between clients and server. The server will take the data whenever it comes from and will possibly interpret them, or to use them to implements some functionalities. Now, if the web application blindly relies on the trust that it has towards its users, bad things may happen. Any data considered by the web application from the user input to be used for any reason, these data needs to be validated before its usage. Unvalidated input from the user may possibly give rise to different kinds of attacks such as XSS, injection flaws and buffer overflow. Validating the input means implementing some functionalities that checks if the input provided by the user is fully compliant with the input that is expected by the application. There are several problems in implementing user input validation : client side or server side input validation ? However, the first approach is completely useless against a dedicated attacker, because he will simply skip them. So, the common case is that we should always implement input validation on server side. In order to proceed with input validation we need to use what is called *positive input validation*. It means that we as software developed should explicitly define the format and limits of the input that we expect. For example we can specify the minimum and maximum input length, the allowed character set, whether null is allowed, specific patterns and so on so forth. A kind of buffer overflow attack that comes from unvalidated input is the **Heartbleed attack** (for more information about that see here).

2.2.1 XSS

Another common kind of attack that comes from unvalidated input is **cross site scripting (XSS)**. XSS is a kind of attack that is strictly related to web applications. The typical case of a XSS attack is the following : there is a web application somewhere, that acquires data from a non-secure source (i.e. user input). The acquired data are sent to other users : they are the target of the attack. Once these users visit the compromised web application, they will receive data from the web application that contains input provided by malicious users. If this input contains client side code, this code is executed on the user browser.



An interesting web site to understand which are the most common attacks against web applications is **OWASP Top 10**. In particular, the information for each attack are divided in three aspects :

- **Attack vectors** : this section contains information about how an adversary may possibly implement the attack
- **Security weakness** : it contains which kind of weaknesses enable this kind of attack and how we can possibly get rid of such weaknesses
- **Impacts** : it contains information about the potential impact of this kind of attack.

A typical way to perform a XSS attack is to embed some user input into a snippet of html code (e.g. inject a script in the $\langle a \rangle$ tag). In particular, there are three common variants of XSS attacks :

- **Stored** : it's a situation where the attack script is already stored somewhere on the website. The attacker uses a vulnerability present in the website, for example a form that doesn't correctly validate the input, to store in a database a malicious script. Then the content of the database is used to serve content to other users, and these users may end up running the malicious script.
- **Reflected** : it's a situation in which the user is pushed by the attacker in making a request to the vulnerable web server, that will send back the attack to the user itself. So the source code of the script is not stored in the server, but is the attacker that through some way (e.g. phishing email) push the user in sending the attack script to the server, and the server reflects back the attack script to the user, that is then compromised.

- **DOM-based** : We may have a vulnerable web page that includes a script that is provided by the web application itself. Some of these scripts may do some use of information that are provided by the user itself. If the input is not appropriately validated, then it can be used to perform a XSS attack. It's called DOM-based attack because the attack is not directly served by the vulnerable web application, but the attack is actually served by the user browser itself that renders the code containing the attack only at the end of the execution of some legit script provided by the web application. A difference with the previous approaches is that the vulnerability is executed on the browser, not anymore on the server.

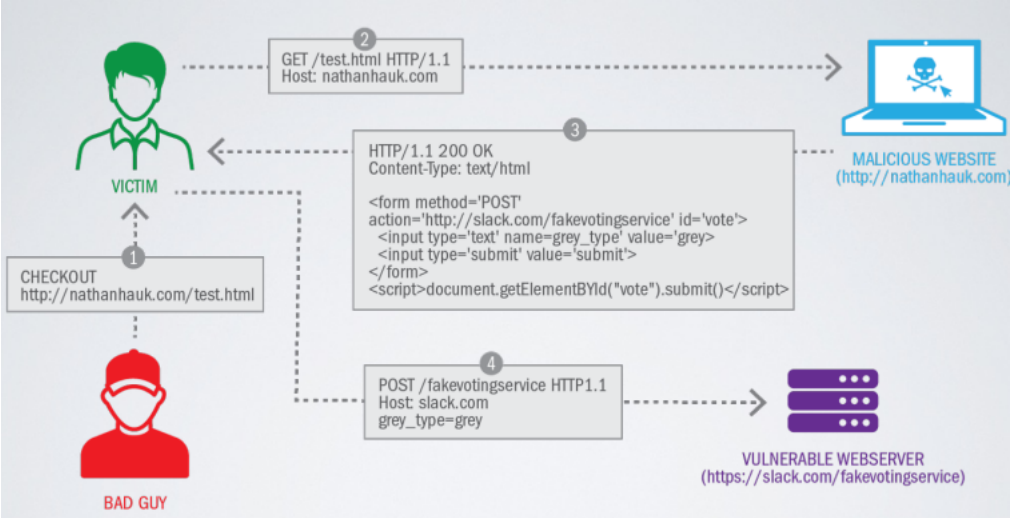
2.2.2 CSRF

Another kind of attack we want to talk about is **Cross Site Request Forgery (CSRF)**. It's similar to XSS attacks, the real difference is that we are not using scripts anymore, but rather we are targeting directly the web application using the user as a vector. The idea of CSRF is that the attacks works on the basis of a simple assumption : the web application is trusting whatever an authenticated user do. The problem is that the attacker with CSRF can push the user to send a request towards the vulnerable web application, without obviously the user being aware of that. The attacker proceeds as follows :

1. First of all the attacker observes how the users interact with the target web application. In particular, he may try to focus its attention on how the users provide data to the web application.

2. Next the attacker builds a fake request that mimics a valid user request. The only problem is that the attacker can't be considered trusted from the web application without a valid session ID.

3. To proceed the attacker lures an authenticated user to visit a website where he has included the form to be submitted (hidden wrt the user). Once the user visit the web page the request will start automatically. Thus, the data submission should be triggered silently and automatically as soon as the user visit that malicious web page.



How does the attacker perform a CSRF attack in practice ? Typically he needs to use a proxy to intercept a non malicious request, in order to understand how the request is performed, because he need to recreate it carefully. Then he setup the malicious web page that contains the CSRF attack with the hidden form and javascript to perform auto submit. Finally, the attacker send the phishing email to the user. The problem is that the target web application blindly trust the requests coming from authenticated users. The problem here is that the web server handles the request even if it's not actually following another request where the user asked the application to provide the form. The solution to this problem is to make sure that all form submissions requires some sort of random token to be submitted every time called **CSRF token**. This token is defined for each user session and be unique, secret and unpredictable. So every time the web application send a form to the user, it will include the CSRF token in the form as a hidden field such that every time the user click submit on the form, the value of the CSRF will return back to the web application and it will be able to check if the content of the token included in the form submission is equal to the token that was defined for that session. If the two are the same token, then it's a request that is coming as a consequence of serving a form to the user. Otherwise it's a potentially CSRF attack.

2.2.3 SQL injection

Injection flaws is a class of vulnerability that allows an attacker to circumvent the functionalities provided by the application and make the appli-

cation execute code that was not in the original design of the application itself. This is the reason why they're called injection flaws, because they allows the attacker to inject its own code in the application and execute the code in the application context. The most common injection attack is called **SQL-injection**, which allows an attacker to inject code in the application in places that actually accepts user input, that will be executed by the interpreter behind the web application itself. In SQL injection attack typically we have that the application on the server makes use of a database back-end to store and retrieve data that is then presented to the user. How does SQL injection works ? The idea is that instead of providing the expected user input to the web application, we provide SQL code to be injected. In fact, the typical scenario, will dynamically creates a SQL query using some data provided externally that can possibly be altered by the attacker, without good validation of such data. Naturally, the target of this attack is always the server that serves the application and the attacker goal is to perform operations that are typically not directly authorized. An important aspect is the fact that if we just consider the case where the attacker wants to inspect and extract data from the database, the problem is that the attacker is limited by how the output of the query is used to craft the interface that is then returned through the web response. In this case the attacker needs to use a different technique that is called **blind SQL injection**. In blind SQL injection the attacker doesn't see directly the result of the query, the only thing that he can see is a binary result (e.g. an error occurred or not). In this case the attacker may ask to the database true/false questions in order to extract 1 bit of information at a time. In order to be robust against SQL injection we need for sure to validate in a proper way the user input, then we can also use additional options such as **parameterized queries** or **stored procedures**. The former are predetermined queries with placeholders that are provided by the runtime environment where the application is actually running. The parameters provided by the user input are used to provide values to those placeholders. The latter are subroutines that are provided by the DBMS itself, which can validate input at server side.

2.2.4 Broken authentication

Authentication being a critical functionality of a web application is typically subject to several kinds of attacks. In particular, it may present vulnerabilities that if correctly exploited by an attacker may allows the attacker to impact the security of the application. Broken authentication methods are mainly based on problems linked to how password-based authentication

is performed and how session security tokens are handled. When dealing with the authentication and the usage of reserved functionalities from users, typically we need to take into account two aspects : the authentication (link between physical and digital identity) and the second one is how do we authorize a specific user represented by a given digital identity to perform or deny the access to some specific functionalities. Password-based authentication has some well known and strong limitations that are mainly related to how password are chosen and managed by the users and website. In particular, attacks to such schema may be related to several aspects of password management. Notice that whatever we do for implementing a correct password management at the server side, the weakness can be on the user side, because most users continue to choose bad password for their digital identities. Predictable passwords are bad because they are an easy target for dictionary-based attacks, reused passwords are even worse, because allows the attacker to perform the so called **credential stuffing attack**, where they have large databases of real password and then try them again and again on real accounts to check if anyone decided to reuse its own password. There are some best practices in how to push users to choose in the right way their passwords. The first practice is pushing people use password managers. Password managers as long as they are used with a local database of passwords, are good solutions for forcing people to use complex passwords because the passwords are generated randomly by the password manager and use different passwords for different services, since the password manager store all passwords for all services and the user needs to remember one single complex password to access the password wallet. The other approach that should be used is the usage of two factor authentication. Using two factor authentication means we challenge the user for at least two authentication methods. Regarding password management at server side we shouldn't store

- passwords as plaintext, because they are easily captured at server compromising
- encrypted password, because if the server is compromised it's likely that the encryption key is also compromised
- secrets (e.g. password digest) deriving from passwords, because digest of dictionary words are easily attacked.

Sessions are subject to attacks through several approaches such as XSS. In fact, in a reflected XSS scenario we trick the user in sending appropriately

crafted javascript code to the web application that reflects it back and make the user browser contact an external service controlled by the attacker and pass to this service information about the session ID. The session ID at that point is linked to an authenticated user and can possibly be used by the attacker to impersonate the user in the vulnerable application. Notice that we can mitigate it in several ways : by setting up the session cookie as a *HttpOnly* cookie the browser will avoid from sending that data together with the other cookie towards the adversary controlled web site. Another approach is to make sure that, during a session, the session ID is linked to a specific IP from which the user was contacting the server. Another kind of attack related to sessions that stems from a wrong implementation of session management is the so called **session fixation** attack. The basic idea is that the vulnerable web site accepts any session identifier. The web application should be designed in a way that it will accept client requests bringing only a session ID that have been issued by the application itself and that it's not expired. This check needs to be done for each single client request. Otherwise the attacker can send an email to the victim, tricking him in contacting the vulnerable web application with a request that already contains a session ID. In this case, if the victim trust the attacker, the user will be challenged for authentication on the vulnerable web application, he will issue its correct credentials and at this point the session ID that have been crafted by the attacker will be associated to a valid identity. To mitigate this kind of attack, the server needs to check that each single session ID that is created, is stored somewhere for its entire lifetime and during its lifetime it can be only associated to a specific user and after its expiry date the session ID should be delete. Furthermore, when a user perform logs out from the web application its session ID should be delete. In such a way, the attacker is not able to reuse session IDs associated to previously logged in users. Another attack is called **session sidejacking**. It's based on the idea that the attacker may possibly setup an attack where he tries to sniff the content of requests between the user and web application. Some sites use TLS encryption for login pages to prevent attackers from seeing the password, but do not use encryption at all for the rest of the site, once the user is authenticated. However, the attacker may sniff the network traffic to intercept all the data that is submitted to the server, including the session cookie. This is one of the reason why today most applications tends to switch to a new model where everything is always served through secured channels. Notice that beside the usage of secure channels, session hijacking attacks can still happen if the user machine has been compromised through a malware, because in that case the malware may possibly tamper the web

browser stealing the session ID from the browser memory space and sending it to the attacker blindly for the user.

2.3 CSP for mitigating XSS

In order to mitigate XSS, we say that this is a consequence of the fact that there is a vulnerability on the server side, because typically user provided content is not correctly validated. Can we do something on the client side ? Yes, we can use CSP to mitigate various types of attacks, including XSS. In practice, CSP is made up by a HTTP response header that instruct the browser on how to make use of the scripts and content that are loaded starting from the web page that is served. In particular, this will limit the kind and origin of scripts that can loaded. CSP is based on the idea that server administrators can through it reduce or possibly eliminate all the vectors through which XSS attacks are executed. In practice, the server provides all content to client that perform the request including a content security policy through which a policy is defined (i.e. *Content-Security-Policy : policy*). In the policy what the administrators does is to specify a set of directives. Typically every policy should include a *default-src* token that identifies the default source that is accepted for content that should be loaded together with the web page (e.g. the *self* value means that, by default, content can be freely loaded by the browser from the domain itself from which the web page was loaded). We can also specify directives for handling different kind of resources such as *img-src* for source of images, *media-src* for source of medias and *script-src* for source of scripts, etc. This is effective because it reduces the options the attacker has to include in a target web page some external content. In addition to whitelisting domains, the policy can also provide two other way to specify trusted resources :

- The CSP directive can specify a nonce and the same value must be used in the tag that loads a script. If these value doesn't match then the script will not be executed.
- The CSP directive can specify a hash of the contents of the trusted script. If the hash of the actual script doesn't match the value specified in the directive, then the script will not be executed.

2.4 Access control attacks

In general access control is a system implementing a methodology that enables some authority to control how users can access specific areas and re-

sources that are made available by an application. In particular, through access control systems we can decide which specific user is allowed to access some specific resource. Notice that access control mechanisms are typically a critical defense mechanism for any applications. When an access control mechanism is defective, typically the attacker can take control of the full application. They are typically tight to the specific applications that are complex to implement and they are often a source of errors. We usually recognize two large families of access controls :

- **Vertical access control** : it allows different types of users to access different parts of the application functionalities. It's commonly used to enforce business policies like separation of duties and least privilege.
- **Horizontal access control** : it allows users to access some subset of resources from a wider range.

In general access control vulnerabilities takes the form of users able to access functionalities even if they are not authorized. We may have two main types of attacks :

- **Vertical privilege escalation** : when a user can perform functions that their assigned role doesn't permit them to do.
- **Horizontal privilege escalation** : when a user can view or modify resources to which he's not entitled.

In particular the access control main weaknesses are divided in the following categories :

- **Completely unprotected functionality** : it's a situation in which sensitive functionalities and resources are partially protected or not protected at all and they can be accessed by knowing the relevant URLs. So we need to take into account that URL can be guessed or brute-forced. Furthermore, links that are created by people accessing with correct credential, will appear in browser history and in the logs of web/proxy servers and they can possibly be read by other people. People can bookmark these URL or email them around.
- **Identifier-based functions** : it's a situation when the identifier that is needed to access a specific functionality is passed to some dynamic web page for example as a parameter. The only thing that the attacker needs to know to access that functionality is the page that provides this resource and the identifier of the resource itself.

- **Multistage functions** : in this case different items of data are captured from the user at each stage (e.g. sequence of forms). This data is strictly checked when first submitted and then is usually passed on to each subsequent stage, using hidden fields. Often developers think that if the user passes the first stage then a further check for that user is not needed since he has already passed the first stage. This is wrong, because the attacker will easily skip the intermediate stage guessing the information that is needed and passed from stage to stage and directly land at the final stage will all the needed information. Another point is that, it's a wrong assumption to think that people will blindly always follows the path that has been provided by the application designers, so that they will always start from the first stage and go down sequentially up to the last stage. The attacker will try all the possible combinations in order to check if by using other paths to access the various stages he may possibly find a way to sidestep the authorization procedure.

- **Static files** : in this case the requests for protected resources are made directly to the static resources themselves, which are located within the web root of the server. At the end we get a static link to a specific resource that is completely unprotected. This means that if an authorized user is able to obtain that link, he will be able to reach the protected resource completely sidestepping all the authentication and authorization procedures. The problem with static resources is that they are not meant to be dynamic code executable by the web server. In some vulnerable applications there aren't effective access control mechanism that place a barrier before accessing that static resources.

2.4.1 Securing access controls

Unfortunately access controls mechanisms are often one of the most vulnerable points in a web application. This stems from several pitfalls that may together act to make our application less secure. First of all a lot of developers tends to implement access control, without properly thinking about the model they need to implement. They have some ignorance about which are the requirements for accessing sensible resources in their application and the lack of a proper knowledge about these requirements is typically a source of vulnerabilities. Furthermore, they typically also make flawed assumptions on the way users may access these resources by performing requests. Another problem is that web application developers often implement access

control mechanisms on a piecemeal basis, simply adding code to individual pages. While implementing properly access control we need to take into account that : we can't trust any user-submitted parameters to signify access rights (e.g. admin = true), we should avoid to assume that users will access application pages in the intended sequence (so implement security mechanisms in depth), we should not trust the user not to tamper with any data that is transmitted via the client. If some data has been validated and is then transmitted through the client, we shouldn't rely upon the retransmitted data without revalidation. First of all access control is not something that we can design while we are implementing the application. We need to properly define, evaluate and document all the requirements linked to access control for every single resource that we want to protect. Naturally we need to take into account least privileges and separation of duties concept in the design phase. All access control decisions cannot be taken on the basis on what the user submit, but only on the basis of the user session. It's a good idea to rely on a central component to check access controls. The advantages of this choice are the following : access control is simplified and easy to understand, the maintenance is more efficient and reliable, improves adaptability and it results in fewer mistakes and omissions. Once we have this central component, we need to make sure that every single user request is authorized by passing through this component. We need to include code that make sure that there aren't exceptions to the previous points. An effective approach from this standpoint, is to make sure that every single functionality is implemented without access control, but the access control is implemented in all pages by intercepting the requests, checking the rights to access such functionalities and only if the central component provides a positive answer then the functionality is executed. If we have extremely sensitive functionality, we can increase the security level by adding further checks such as restrict access by IP address to ensure that only users from a specific network range are able to access the functionality. Access control wrt static content can be implemented through a few methodologies. In the most simple case we can protect this content through basic HTTP authentication and then provide user with the credential to access this resource. The other approach is to mask the location of these resources through a properly designed dynamic server side page. We link the server side page through which we pass a parameter, and this parameter is then mapped to the real resource that is only accessed by the dynamic web page that then reroutes the content to the user.

3 Tor

The Internet was designed as a public network and everything we do on the web is tracked and logged in general. Nevertheless today we have a lot of tools that we use everyday and that scale perfectly to guarantee the security of our interactions with a service (e.g. encryption). However, these tools doesn't do anything for guaranteeing our anonymity. If we setup a HTTPS channel, and someone places itself in the middle trying to check the content of the channel, he will not be able to look at the content, but he will perfectly aware of the fact that we are trying to communicate with a specific website at a specific point in time. He will able also to track statistical information such as how many times we connect to such website, average session duration, etc. For anonymity we means that whatever we do on the web, we should not be identifiable as the source or the destination of that activity. It stems from two different aspects :

- **Unlinkability** of actions and identity : if someone observes what we are doing, he is not be able to understand what we are doing.
- **Unobservability** : even if an adversary is observing what we are doing, he has no way to understand which system or protocol we are using.

The attacker to break unlinkability requirement can possibly perform passive or active traffic analysis. The first one means trying to look at packet flow on a network link, and try to infer who is talking whom. Active traffic analysis where the attacker is able to actively inject data in the system and may use the effect of these network data to try to identify someone or some service. We also assume that, the attacker in order to perform its analysis is able to compromise some network nodes such as routers, firewalls, etc. The idea is that we can't trust the underlying network on top of which the Internet is based for anonymity. Actually we need to build something on top of the network that allows us to overcome all the threats paused by the previous threat model and still guarantee unlinkability between users while they communicate. There are several proposals dedicated for implementing anonymous services for end users, **The Onion Router (Tor)** is by far the most successful among all these projects. TOR is a service that we run on our device and other software can use to route traffic in anonymous way towards the desired destination. In general, TOR is distributed anonymous overlay network that allows participants to exchange information in a completely anonymous fashion. An interesting point is that, thanks to its architecture,

TOR allows its users also to access open standard web services. In this way the client is completely anonymized wrt the server. TOR offers another kind of functionality that goes under the name of **hidden services** that also implement the other side of anonymity. It allows users to access in an anonymous fashion services that are by themselves anonymous. In this case both the client and server identity are not revealed to the other endpoint of the channel.

3.1 Architecture

In Tor we have the following actors :

- **Client** : the user of the Tor network
- **Server** : the target TCP application such as web servers
- **Tor router** : the special proxy relays the application data
- **Directory server** : they are servers holding Tor router information.

How Tor works ? Assume that Alice is trying to contact Bob to obtain a service from him. To do that, the first thing that Alice needs to do is to create a protected, encrypted and anonymous link to Bob using the Tor network. In particular, she contact Dave, which represent one of the Tor directory server, to obtain a link to a certain number of Tor relays. A Tor relay is a Tor service running on the pc of someone on the web. At this point Alice will receive the identity of 3 relays (we assumed assumed such number for simplicity) and use them to create an encrypted link, where each of them represent an intermediate hop before contacting Bob. In other words, Alice is creating a virtual circuit within the Tor network, that will make her exchange data with Bob, but indirectly. Alice will be anonymous wrt Bob, because her packets will be indirectly received by Bob through the last Tor relay in the virtual circuit setup by Alice. So why the directory server provides Alice the identity of 3 Tor relays ? Because virtual circuits in Tor are made up of three relay nodes. The first node is called **guard relay**, a middle relay and an exit relay. The guard relay is the first Tor relay that is contacted directly by Alice to send data towards Bob. The exit relay is the last relay node in the virtual circuit, that has the role to interact directly with Bob. The middle relay is in charge to decouple the interactions between the guard relay and exit relay. Even within the Tor relay network, connections in a virtual circuit are performed through intermediate nodes. This approach is completely different from the standard

approach used by VPNs. Suppose Alice connect to the VPN using some protocol such as IPSec, and this means that we are setting up a virtual channel that is encrypted between us and the VPN server that is provided by our VPN provider. Next, the VPN server will immediately contact Bob (there will be a single indirection node between the two endpoints). This is a problem for the anonymity endpoint, because while Bob still sees packets coming from a VPN server and not directly from Alice, the VPN server itself has full visibility of what happens between Alice and Bob. Thus the unlinkability property that is needed for anonymity is not satisfied by VPN services. The usage of three intermediate relays in the construction of the Tor virtual private channel, is exactly what is needed by Tor to guarantee unlinkability. Notice that, a virtual circuit typically tend to last the whole session of an interaction between Alice and Bob. Thus, the traffic between Alice and Jane, will pass over a completely different and uncorrelated channel wrt the channel between Alice and Bob. This is needed to avoid the Tor network itself to be able to snoop upon Alice interactions with external services. The magic of unlinkability is provided within a virtual channel by a routing approach called **Onion Routing**. The basic concept is that Alice will send data to its endpoint using the virtual channel, but will encrypt data whose destination is Bob using a specific encryption approach called **Onion-like encryption**. In particular, to send data Alice encrypts data applying several layers of encryption that are setup such that each layer can be removed by a single onion relay. The data will first be encrypted using a symmetric key shared between Alice and the exit relay of the virtual circuit. Next, the data is encrypted again with a symmetric key shared between Alice and the middle relay of the circuit. Then, the data is encrypted again using a symmetric key shared between Alice and the guard relay.

Tor circuit setup First of all the client establish a symmetric key with the guard relay and to start the creation of a virtual circuit. In doing so the client exposes its identity to the guard relay. Thus the guard relay will act as an intermediate actor between the client and the middle relay in the virtual circuit. In particular, the client receives from the directory server together with the identity of the relays some information, like their IP address, but also their public key. At this point the client uses this public key to encrypt a session key that is sent through the guard relay towards the middle relay. The middle relay receives a request to setup the virtual circuit and the guard relay will include in such request also a virtual circuit identifier. Now, the middle relay will setup the data structures associated

to this new virtual circuit and will decrypt the received session key, and will use it to create a symmetric key that will be used to exchange data with the client. The symmetric key is encrypted with the session key sent by the client, and it will send back to the client passing through the guard relay. In this way, the client and the middle relay can setup their shared key in such a way that the identity of the client is never disclosed to the middle relay. The same approach is used to setup the symmetric key with the exit relay, but the requests will be related by the guard and middle relay towards the exit relay. At the end of this protocol, all three relays will share a symmetric key with the client that is know only by the client and the specific relay node. Once the virtual circuit is created, then it can be used to contact other external services using the symmetric keys for encryption. In order to deanonymize the connection made by a user towards a specific endpoint, an attacker should control all the three relays in the virtual circuit.

The design of Tor is made up of several components that collaborate to implement the functionalities that we have seen so far. At the first level we have the overlay network which is constituted by virtual routers that works at user level, and that interconnect and communicate with other routers such to realize a network on top of another network. It's important to known that onion routers are implemented through software that normal people execute on their own computers or servers and it's made up to work with standard user privileges. Then we have the onion proxy, which is the software needed to be installed by users in order to connect to the Tor overlay network and make use of its functionalities. Tor for implementing at network level the way routers and proxy exchange information is by using TCP with TLS in order to avoid Tor traffic to be easily filtered on firewalls (in this way Tor hides that we are using it). Another characteristics of the network data flow generated by Tor is that all data sent by Tor is sent in fixed size cells in order to reducing the amount of information that can be get by looking at the size of the packets. However, the packet frequency is not always the same, since the exit relay of the virtual circuit adds some noise to the frequency of packets that are sent toward the endpoint. Other functionalities that are enforced by Tor design are the following : all packets are checked for integrity, there is a mechanism called **link throttling** to avoid congestion on links (especially for links that interconnects onion routers). There is also a form of traffic throttling that is applied at the level of virtual circuit; if in the virtual circuit there are too much requests going in one direction without enough answers coming back from the endpoint of the connection, further requests will be throttled waiting for the response to come back. However,

the problem of having a public list of onion routers is that they can be all filtered out, avoiding the creation of a virtual circuit at all. To overcome this limitation the Tor project also keeps a sort of secret list of alternative onion routers that are called **onion bridges**. The directory servers doesn't known the onion bridges, since they are not part of the public list of onion routers. If we live in place where we cannot connect to Tor, because the connection to standard onion routers is filtered, we may ask the Tor project to provide us with a link with the identity of an onion bridge. The bridge acts as an intermediate point between the proxy and the guard router in the virtual circuit. The bridge will create virtual circuits on behalf of the proxy that is behind a firewall.

3.2 Hidden services

Hidden services are Internet services that are accessible only through Tor overlay network, they are not visible or usable through the standard Internet. The idea of making a service available through Tor is that the owner and administrators of those services have their identity anonymized. Anyone can connect to this website without knowing who is running that website. The original idea behind hidden services was to create something that was resistant to censorship. Setting up a hidden service is similar to setting up a virtual circuit for a client that wants to exchange data with an external service. In this case, it makes sense that both client and the service has their own virtual circuit, because in this case we want to ensure the anonymity for both of them. Now, we need to understand how Tor make sure that these virtual circuits have a point in common, that is an onion router where the two circuits collapse on the same point.

Hidden service setup Suppose that Bob wants to expose a hidden service. The first step to provide such service is that the proxy on Bob side selects a certain number (typically 3) of onion routers chosen at random. These onion routers are called **introduction points**. These introduction points are contacted by Bob by setting up three independent virtual circuits. Each introduction point doesn't know the identity of Bob. Then Bob locally creates a sort of service identity token that contains several information such service name, unique identifier of the service, list of introductions points and the service public key. Notice that the identity of a hidden service typically in Tor takes this form *service_name.onion*. This token is then published by Bob in a public database. This database is a single place where users can possibly query for information about available hidden services. If Alice

wants to access the hidden service, she queries the database for the information token about the service. Then she selects at random an onion router that will be called the **rendezvous point** for the connection between Alice and Bob. Now the problem is that, Alice needs a way to say to Bob in an anonymous manner, let's talk using that onion router. For this reason, Alice creates a connection request token, that is made up by setting up a piece of information that contains a cookie (a random number generated by Alice) and the identity of the rendezvous point selected by her. All these information are encrypted using the hidden service public key, which was contained in the information obtained by querying the database. Alice takes this encrypted request and creates a virtual circuit toward one of the three introduction points previously chosen by Bob (list of introduction points obtained again from the database). Next, she send to the chosen introduction point this request token, asking to it to route the request toward the hidden service. Once Bob obtains the request from Alice, he will decrypt the request using its private key. First of all he checks the identity of the rendezvous point. If for some reason Bob decide that some rendezvous points are not eligible for setting up connections, he will simply drop the request. Alice takes a TTL on the request, if it's not satisfied within a certain amount of time by Bob, she will try to make a new request selecting a new rendezvous point. If the rendezvous point chosen by Alice is ok for Bob, then Bob will create a virtual circuit towards the rendezvous point providing as information the cookie. At the same time also Alice creates a virtual circuit towards the rendezvous point, and place on it a connection token that contains the same cookie. This is the way the rendezvous point will be able to match two different virtual channels one coming from Alice, the other one coming from the hidden service, that both contains the same cookie. Once the rendezvous point sees the same cookie coming from the two virtual channels, it will connect them and act as a router between them. This means that all data that needs to go from Alice to the hidden service will pass through the rendezvous point. In particular, starting from Alice the data will reach the rendezvous point, and then proceed on the virtual circuit between this latter and the hidden service. By using two independent virtual circuits, we guarantee that the rendezvous point will not know anything about the identity of both Alice and Bob (the hidden service itself). At the same time we guarantee that Bob will not know nothing the identity of Alice and Alice will not know anything about the identity of Bob.

Since the database represent a single point of failure, it's implemented in Tor by the whole community of onion routers through what is called **dis-**

tributed hash table. It works exactly as a hash table, the only difference is that it's implemented through different machines, where each of them stores a subset of all the possible keys. Then it's implemented through a distributed protocol that enables us to access every key-value pair that is stored, by simply querying one node that makes up this distributed hash table, and the internal protocol will make sure that, by forwarding our request through several nodes, we will able to reach the node that actually keeps in memory that key-value pair that we are looking for. Furthermore, this kind of data structure implemented mechanisms that allow to replicate data such that if a node that stores a portion of data of the table fails, the data will be replicated on a different node. They also allow for a dynamic implementation of the hash table. Nodes that implement the distributed hash table can enter or leave the overlay network, and data will not be lost. When a node enters the overlay network a portion of existing data will be moved on this node. When a node leave the network the data that it handles will be moved to another node.